CMPSC 274: Transaction Processing
Lecture #6: Concurrency Control Protocols

Divy Agrawal
Department of Computer Science
UC Santa Barbara

Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- 4.3 Locking Schedulers
- 4.4 Non-Locking Schedulers
  - 4.4.1 Timestamp Ordering
  - 4.4.2 Serialization Graph Testing
  - 4.4.3 Optimistic Protocols
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned
(Basic) Timestamp Ordering

**Timestamp ordering rule (TO rule):**
Each transaction $t_i$ is assigned a unique timestamp $ts(t_i)$ (e.g., the time of $t_i$’s beginning).
If $p_i(x)$ and $q_j(x)$ are in conflict, then the following must hold:
$p_i(x) < q_j(x)$ iff $ts(t_i) < ts(t_j)$ for every schedule $s$.

**Theorem 4.15:**
Gen (TO) $\subseteq$ CSR.

**Basic timestamp ordering protocol (BTO):**
- For each data item $x$ maintain $max-r(x) = \max\{ts(t_j) \mid r_j(x)$ has been scheduled} and $max-w(x) = \max\{ts(t_j) \mid w_j(x)$ has been scheduled}.
- Operation $p_i(x)$ is compared to $max-q(x)$ for each conflicting $q$:
  - if $ts(t_i) < max-q(x)$ for some $q$ then abort $t_i$
  - else schedule $p_i(x)$ for execution and set $max-p(x)$ to $ts(t_i)$

---

**BTO Example**

$s = r_1(x) \; w_2(x) \; r_3(y) \; w_2(y) \; c_2 \; w_3(z) \; c_3 \; r_1(z) \; c_1$

\[ r_1(x) \; w_2(x) \; r_3(y) \; a_2 \; w_3(z) \; c_3 \; a_1 \]
Chapter 4: Concurrency Control Algorithms

• 4.2 General Scheduler Design
• 4.3 Locking Schedulers
• 4.4 Non-Locking Schedulers
  • 4.4.1 Timestamp Ordering
  • 4.4.2 Serialization-Graph Testing
• 4.4.3 Optimistic Protocols
• 4.5 Hybrid Protocols
• 4.6 Lessons Learned

Serialization Graph Testing (SGT)

SGT protocol:
• For \( p_i(x) \) create a new node in the graph if it is the first operation of \( t_i \).
• Insert edges \((t_j, t_i)\) for each \( q_j(x) < p(x) \) that is in conflict with \( p_i(x) \) (i.e., \( i \neq j \)).
• If the graph has become cyclic then abort \( t_i \) (and remove it from the graph) else schedule \( p_i(x) \) for execution.

Theorem 4.16:
Gen (SGT) = CSR.

Node deletion rule:
A node \( t_i \) in the graph (and its incident edges) can be removed when it is terminated and is a source node (i.e., has no incoming edges).

Example:
\( r_1(x) \, w_2(x) \, w_2(y) \, c_2 \, r_1(y) \, c_3 \)
removing node \( t_2 \) at the time of \( c_2 \)
would make it impossible to detect the cycle.
Chapter 4: Concurrency Control Algorithms

• 4.2 General Scheduler Design
• 4.3 Locking Schedulers
• 4.4 Non-Locking Schedulers
  • 4.4.1 Timestamp Ordering
  • 4.4.2 Serialization-Graph Testing
• 4.4.3 Optimistic Protocols
• 4.5 Hybrid Protocols
• 4.6 Lessons Learned

Optimistic Protocols

Motivation: conflicts are infrequent

Approach:
divide each transaction t into three phases:
  read phase:
    execute transaction with writes into private workspace
  validation phase (certifier):
    upon t’s commit request
    test if schedule remains CSR if t is committed now
    based on t’s read set RS(t) and write set WS(t)
  write phase:
    upon successful validation
    transfer the workspace contents into the database
    (deferred writes)
    otherwise abort t (i.e., discard workspace)
Backward-oriented Optimistic CC (BOCC)

Execute a transaction’s validation and write phase together as a critical section:
while $t_j$ being in the val-write phase, no other $t_k$ can enter its val-write phase

**BOCC validation** of $t_j$:
compare $t_j$ to all previously committed $t_i$,
accept $t_j$ if one of the following holds
• $t_i$ has ended before $t_j$ has started, or
• $RS(t_j) \cap WS(t_i) = \emptyset$ and $t_i$ has validated before $t_j$

**Theorem 4.46:**
Gen (BOCC) $\subseteq$ CSR.

**Proof:**
Assume that $G(s)$ is acyclic. Adding a newly validated transaction can insert only edges into the new node, but no outgoing edges (i.e., the new node is last in the serialization order).

BOCC Example
Forward-oriented Optimistic CC (FOCC)

Execute a transaction’s val-write phase as a strong critical section:
while \( t_i \) being in the val-write phase, no other \( t_k \) can perform any steps.

**FOCC validation** of \( t_j \):
compare \( t_j \) to all concurrently active \( t_i \) (which must be in their read phase)
accept \( t_j \) if \( WS(t_j) \cap RS^*(t_i) = \emptyset \) where \( RS^*(t_i) \) is the current read set of \( t_i \)

Remarks:
• FOCC is much more flexible than BOCC:
  upon unsuccessful validation of \( t_j \) it has three options:
  • abort \( t_j \)
  • abort one of the active \( t_i \) for which \( RS^*(t_i) \) and \( WS(t_j) \) intersect
  • wait and retry the validation of \( t_j \) later
    (after the commit of the intersecting \( t_i \))
  • Read-only transactions do not need to validate at all.

Correctness of FOCC

**Theorem 4.18:**
\( \text{Gen (FOCC)} \subseteq \text{CSR} \).

**Proof:**
Assume that \( G(s) \) has been acyclic and that validating \( t_j \) would create a cycle.
So \( t_j \) would have to have an outgoing edge to an already committed \( t_k \).
However, for all previously committed \( t_k \) the following holds:
• If \( t_k \) was committed before \( t_j \) started, then no edge \((t_j, t_k)\) is possible.
• If \( t_k \) was in its read phase when \( t_j \) validated, then \( WS(t_k) \) must be
disjoint with \( RS^*(t_j) \) and all later reads of \( t_j \) and all writes of \( t_j \)
must follow \( t_k \) (because of the strong critical section);
so neither a wr nor a ww/rw edge \((t_j, t_k)\) is possible.
Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- 4.3 Locking Schedulers
- 4.4 Non-Locking Schedulers
- **4.5 Hybrid Protocols**
- 4.6 Lessons Learned
Hybrid Protocols

Idea: Combine different protocols, each handling different types of conflicts (rw/wr vs. ww) or data partitions

Caveat: The combination must guarantee that the union of the underlying “local” conflict graphs is acyclic.

Example 4.15:
use SS2PL for rw/wr synchronization and TO or TWR for ww with TWR (Thomas’ write rule) as follows:
for \( w_i(x) \): if \( ts(t_i) > \text{max-w}(x) \) then execute \( w_i(x) \) else do nothing

\[
\begin{align*}
    s_1 &= w_1(x) \ r_2(y) \ w_2(x) \ w_2(y) \ c_2 \ w_1(y) \ c_1 \\
    s_2 &= w_1(x) \ r_2(y) \ w_2(x) \ w_2(y) \ c_2 \ r_2(y) \ w_2(y) \ c_1
\end{align*}
\]

both accepted by SS2PL/TWR with \( ts(t_1) < ts(t_2) \), but \( s_2 \) is not CSR

Problem with \( s_2 \): needs synch among the two “local” serialization orders

Solution: assign timestamps such that the serialization orders of SS2PL and TWR are in line

\[ ts(i) < ts(j) \Leftrightarrow c_i < c_j \]

Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- 4.3 Locking Schedulers
- 4.4 Non-Locking Schedulers
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned
Lessons Learned

• S2PL is the most versatile and robust protocol and widely used in practice
• Knowledge about specifically restricted access patterns facilitates non-two-phase locking protocols (e.g., TL, AL)
• O2PL and SGT are more powerful but have more overhead
• FOCC can be attractive for specific workloads
• Hybrid protocols are conceivable but non-trivial